第5章 SystemVerilog アサーション入門(前編)

内部信号のふるまいをツールが自動監視するデバッグ手法

赤星博輝

SystemVerilogの新しい特徴の一つとしてアサーション(assertion;「表明」,「主張」を意味することば)への対応がある.アサーションでは,あらかじめ内部信号のふるまいを定義し,回路がそのとおりに動作しているかどうかをシミュレータに自動監視させる.デバッグの効率化や検証環境の再利用などに効果がある.また,アサーションの記述は形式的検証ツールの一種であるプロパティ検証ツールの入力データとしても利用できる.(編集部)

SystemVerilogでは,これまでVerilog HDLが弱かった検証機能を大幅に強化しました.例えば,制約付きランダム・テスト生成,機能カバレッジ,インターフェース,アサーションなどの機能が追加されています(p.84のコラム「SystemVerilogのここが良い,ここが悪い」を参照).

現在のLSI設計は,**図1**のように設計チームと検証チームが共同でボートをこぐ状況にたとえることができます.検証側が右側を,設計側が左側をこぐとすると,短時間でゴールするためには,検証と設計のこぎ手が協調してバランスよくこぐ必要があります.もし,右側と左側のこぎ手のバランスがとれていないとボートはまっすぐに進みません.現在は,どちらかというと設計に比重が置かれているため,ボートはまっすぐに進まず,蛇行しがちに見えます.

もし,検証力を高めることができれば,設計期間だけでなく設計品質などにも良い影響を与えることでしょう.しかし,ただ「検証力を強化しろ」と言われても,現実にどうやってよいのかと悩むかと思います.検証のレベル・アップに特効薬はなく,ひとつひとつ積み上げていくしかないと考えます.まず何をやってよいのかわからないときには,ここでご紹介する「アサーション」を導入してみることをお勧めします.

● アサーションを使う四つのメリット

LSIを設計するとき,設計した回路が正しく仕様を満たしているかどうかをチェックする必要があります.例えばカウンタを設計している場合,1サイクル進むと値がインクリメント(+1)されているかどうかをチェックする必要があります.このチェック項目が,カウンタという回路が満たすべき条件(仕様)になります.このような条件を「プロパティ」と呼び,このプロパティを検証言語(アサーション言語)などで記述したものをここでは「アサーション」と呼ぶことにします(p.86のコラム「SystemVerilogアサーションとPSLの比較」を参照).

検証で重要なことは,このチェック項目をはっきりさせることです.アサーションの導入でうまくいかない原因の一つに,チェック項目がはっきりしないためにアサーションが書けない,ということがあります.そのため,「アサーションを書く」という作業を通じて,設計や検証のやりかたを見直すことが重要です.



図1 設計と検証はバランスが重要
検証期間が延びているのは,設計が複雑になったからというよりも,設計と検証のバランスが悪くなったからだろう.設計者と検証エンジニアが力を合わせて,最速かつ最短時間でゴールに着きたいものである.

Column　SystemVerilogのここが良い，ここが悪い

筆者がSystemVerilogに注目し始めたのは2002年の初めごろです．世の中はC言語をベースとしたシステムLSI設計手法が立ち上がろうとしていたときですが，どうしてもCベース設計に向いていない領域があることを感じていました．その一方で，SystemVerilogは，ハードウェア設計者にとってたいへん魅力のある言語と感じられました．

SystemVerilogの言語仕様はどんどん改訂されています．Version 3.0からVersion 3.1aになり，LRM（language reference manual）は140ページから584ページへと厚くなりました．筆者が満足している点は，これまでVerilog HDLの弱かったところや問題があったところが大幅に改善されたことです．これにより，いろいろな言語を併用する必要がなくなり，実質的にはSystemVerilogとC言語でほとんどのことができるようになりました（とはいえ，C言語はやっぱり必要なのだが…）．

不満な点については，実はあまりないのですが，SystemVerilogで追加された機能の多くはこれまでほかの言語（検証言語など）でサポートされてきたものであり，それほど革新的な機能は含まれていません．設計者の側の習得の努力も必要ではありますが，EDAツールや言語の側についてもさらなる進歩を期待します．

また，日本語（おそらく英語でもそうなのだが…）で書いた文章にはあいまいさが残る場合があります．例えば，「処理Aから3クロック以内に処理Bを実行する」という仕様は，設計グループ内で誤解なく理解できるでしょうか？ 文章だけではできないはずです．例えば，「処理Aが開始されてから3クロック以内に処理Bが完了する」と解釈する人もいれば，「処理Aが完了して3クロック以内に処理Bを開始する」と解釈する人もいるかもしれません．これで誤解が発生しない場合は，その設計グループの中に暗黙の了解事項があるということになります．こうした暗黙の了解事項の量が多いと，新しいメンバが入ってきたときに問題が発生しやすくなります．このときに，検証言語などで仕様を記述していれば，仕様を明確に伝えられ，さらに，その仕様をツールによってチェックできます．

昨今の検証がたいへんなのは，設計する回路の規模が大きくかつ複雑になっていることが原因です．設計サイズが大きくなると，与えた入力に対して出力ポートの値を観測するだけでは，バグを発見できる可能性が少なくなってきています．せっかく内部でバグが発生する入力パターンを与えても，バグの値が出力ポートに出てこなければ，バグを発見できないからです．

もちろん気長に出力ポートまで値を導き出すテストベンチを作成すればよいのですが，検証をすばやく完了するためには，内部の値を確認することが必要になります．また，大規模な設計では，出力ポートの値がまちがっていた場合にどこが原因なのかを突き止めるデバッグ作業が必要になり，内部の信号を確認しながらバグの原因を特定していきます．

また，検証の課題の一つとして，作業効率の問題が挙げられます．例えば，シミュレーション結果を波形で確認する作業を行っている場合がありますが，ほんとうに人が画面に張り付いて，目視で確認する必要があるのでしょうか？ 問題点の一つは，人間はミスをするということです．大規模な設計では，人為的なミスが入る可能性をできる限り減らす必要があります．また，設計修正やシミュレーション・パターンの追加が生じることは珍しくないのですが，そのたびに設計者が波形を確認していたのでは，効率が悪すぎます．

アサーションを使うことによるメリットには，検証項目の明確化，検証環境の再利用，内部信号の観測によるバグの早期発見，検証の効率化といった四つのポイントがあります．今回は，このアサーションをシミュレータで利用する方法を中心に説明します．

● 即時アサーションはinitialなどのブロックで使う

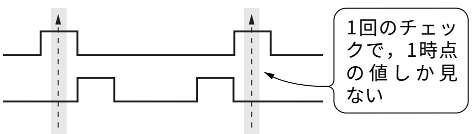
SystemVerilogでは，即時アサーション（immediate assertion）と並列アサーション（concurrent assertion）の2種類のアサーションが導入されました．この二つの違いですが，**図2**に示すようにある時点の信号の値だけを見るのが即時アサーションで，複数サイクルにまたがる信号の値を見ることができるのが並列アサーションになります．

まず，即時アサーションについて説明します．

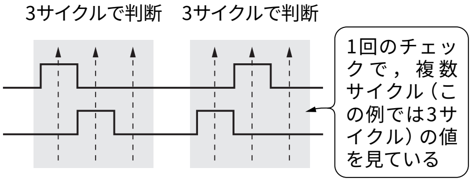
即時アサーションは，呼び出されたときにプロパティを満たしているかどうかをチェックします．構文は，以下のようになります．

```
assert ( <論理式> ) [<成立時に実行する文>]
                 [else <不成立時に実行する文>];
```

呼び出されたときにチェックするので，いつ呼び出すかを`initial`ブロックや`always`ブロックで記述する必要



（a）即時アサーションのイメージ



（b）並列アサーションのイメージ

図2　即時アサーションと並列アサーションの違い
即時アサーションはある1時点の値だけを見て判断する．並列アサーションは複数サイクルの値を見て判断する．プロトコルなどのチェックでは，並列アサーションが必要になる．

リスト1　即時アサーション

```
initial begin
#100 assert ( count < 4 ) $display(" <4 "); else $display (" >=4");
end
```

（a）initial ブロック

```
always @( count ) begin
  if ( count < 4 ) less4 = 1'b1;
  else             less4 = 1'b0;
  assert ( count < 4 ) $display(" <4 "); else $display (" >=4");
end
```

（b）always ブロック

があります（そのほか，SystemVerilogの interface や program でも使用可能）．initial ブロックや always ブロックの記述例を見ていきます．

まず，initial 中に記述した例を**リスト1（a）**に示します．100 遅延の後に1度だけ assert が呼び出され，そのときの count の値によって出力メッセージを決定します．

次に，always 中に記述した例を**リスト1（b）**に示します．これは，組み合わせ回路を生成する always の中にアサーションを埋め込んでいるので，count の値に変化があれば，このアサーションが呼び出されることになります．

即時アサーションは，initial や always などのブロック内で，実行する条件が整った時点で実行されることが特徴です．

● 並列アサーションで複数サイクルの動作をチェック

並列アサーションは，複数サイクルに対してチェックを行うものです．現在，検証手法としてよく話題になっているアサーションは，おもに並列アサーションです．回路の検証では，ある時点だけで正しい設計かどうかを見極めることは困難で，多くの場合，複数サイクルにわたって回路の動きをチェックする必要があります．アサーションを使わずにこのような仕様を検証しようとすると，設計者が波形を確認しなければならず，どうしても時間がかかります．そのような場合，並列アサーションが使えないかどうかを考えてみましょう．

並列アサーションは，複数サイクルにまたがった動作を表現するため，即時アサーションより複雑な記述になりがちです．並列アサーションは，以下のように分類できます．

- **複数サイクルのプロパティ**
- **条件があるプロパティ**
- **繰り返し記述**
- **アサーションのキャンセル**
- **再利用のための記述（プロパティ定義，シーケンス定義）**
- **プロパティ記述で有効な演算**
- **あらかじめ用意された関数**

「なぜこれらの記述方法を覚えないといけないのか？」ということですが，アサーションはいわば"正解の波形"を記述するものです．検証ではその正解の波形が書けないと話になりませんし，正解の波形を書くのに手間や時間がかかってもいけません．そのために，こうした記述を覚える必要があるのです．

● クロック指定やサイクル遅延を組み合わせる

即時アサーションはある1時点のプロパティを記述するので，記述上は時間に関して考慮する必要がありません．一方，並列アサーションでは複数サイクルにまたがるプロパティを記述するため，時間も含めて表現する必要があります．例えば，「最初のサイクルで1ビットの信号 a が'1'で，次のサイクルで1ビットの信号 b が'1'になる」をプロパティとして書くことから始めます．

SystemVerilog ではサイクル遅延というものが導入されました．これまで Verilog HDL では #10 などとして遅延時間を埋め込んでいましたが，SystemVerilog では ##1 と記述することで1サイクルの遅延，##2 とすることで2サイクルの遅延を表現できます．

これにより，a ##1 bと書くことで，「1ビットの信号aが‘1’で，次のサイクルで1ビットの信号bが‘1’になる」を表現できます．それではこのサイクルとは何を基準にするのでしょう？ 並列アサーションでは，どこかでこのサイクルの基準となるものを定義する必要があります．同期設計を行っている場合は，クロックの定義を使うことができます．クロックの指定方法はVerilog HDLと同じで，@(posedge clk)や@(negedge clk)として基準となるタイミングを指定します．

仕様を並列アサーションとして記述するには，assert propertyというキーワードを使って以下のように定義します．

```
assert property ( <クロック指定> <プロパティ> )
        [<成立時の文>] [else <不成立時の文>];
```

これで，記述するための準備はできました．それでは，「基準となるクロックは信号clkの立ち上がり」，「最初のサイクルで1ビットの信号aが‘1’で，次のサイクルで1ビットの信号bが‘1’になる」というアサーションを以下に示します．成立時にはOK，不成立時にはNGを出力するようにしています．

```
assert property ( @(posedge clk) a ##1 b
) $display("OK"); else $display("NG");
```

ここでは，OKやNGのメッセージを表示するアサーションにしましたが，このようなメッセージはシミュレータが出力してくれるので，わざわざ記述する必要はありません．チェックするだけなら成立時や不成立時にメッセージを出力する部分はなくてもかまわず，以下の記述で十分です．

```
assert property
        ( @(posedge clk) a ##1 b );
```

このアサーションがどのようにチェックされるかを示したのが**図3**です．並列アサーションでは，指定したクロックのイベントが発生すると，チェックを開始します．時刻T0で開始されたアサーションは信号aが‘0’のためその時点でFAILですが，時刻T1で開始されたアサーションは信号aが‘1’のためチェックを継続し，時刻T2で信号bが‘1’のためPASSします．

このa ##1 bでチェックされるのは，**図4**に示した4通りの可能性があります．もし，この四つを区別したければ，それぞれのふるまいを正確に記述する必要があります．

即時アサーションは，明示的に呼んだときにチェックしていましたが，この並列アサーションは複数サイクルにわたってチェックするため，記述する場所が異なります．ここではmodule内に記述する方法を**リスト2**に示します．alwaysやinitialと同じレベルで記述できます．

● 条件付きの場合は|->，|=>を使用

回路の中には，つねにチェックしなければならない仕様だけでなく，ある条件が成り立ったときだけチェックすべ

Column SystemVerilogアサーションとPSLの比較

SystemVerilogアサーションとPSL（Property Specification Language）の比較表を作成してみました（**表A-1**）．どちらもアサーションを記述できる言語です．完全に一致しないものもありますが，PSLを使っている方には参考になるかと思います．

表A-1 SystemVerilogアサーションとPSL

項目	SystemVerilog 3.1a	PSL 1.1
ワンホットの判定	$onehot	onehot
ワンホットかまたはALL0かの判定	$onehot0	onehot0
不定を含むかの判定	$isunknown	isunknown
2進数における‘1’の数を数える	$countones	countones
アサーションの値を観測	$sampled	―
立ち上がりの判定	$rose	rose
立ち下がりの判定	$fell	fell
無変化の判定	$stable	stable
過去の値の参照	$past	prev
将来の値の参照	―	next

（a）関数

項目	SystemVerilog 3.1a	PSL 1.1
シーケンスの連結	s1 ##1 s2	s1 ; s2
シーケンスの融合	s1 ##0 s2	s1 : s2
シーケンスの長さ不一致and	s1 and s2	s1 & s2
シーケンスの長さ一致and	s1 intersect s2	s1 && s2
シーケンスのor	s1 or s2	s1 | s2
シーケンスのwithin	s1 within s2	s1 within s2
最初の成立したシーケンス	first_match(s1)	―
連続繰返し	a[*n:$]	a[*n:inf]
GOTO繰り返し	a[->n]	a[->n]
非連続繰り返し	a[=n]	a[=n]
アサーションのキャンセル	disable iff	abort

（b）シーケンス，繰り返し

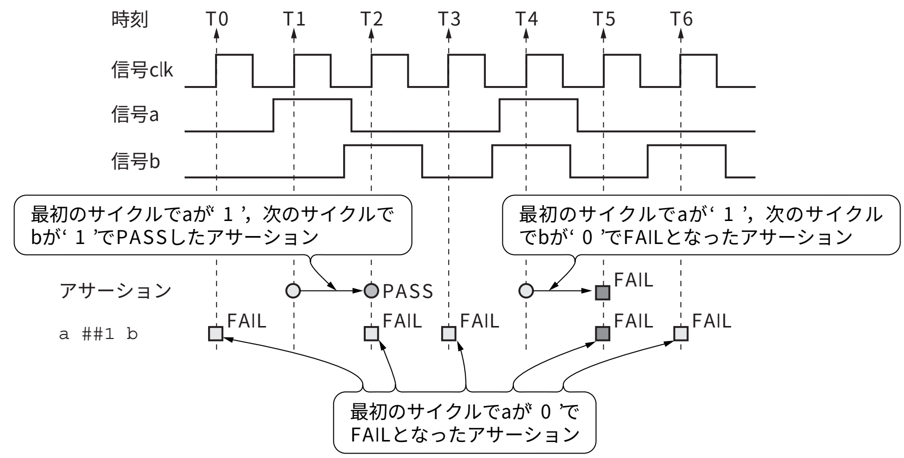


図3
アサーションのふるまい
アサーションはサイクルごとにチェックする．このアサーションをすべての時刻において成立させるには，信号`a`はつねに'1'で，信号`b`は2サイクル目からつねに'1'になる必要がある．複数サイクルにまたがるアサーションでは，なんらかの条件が付くことが多い．

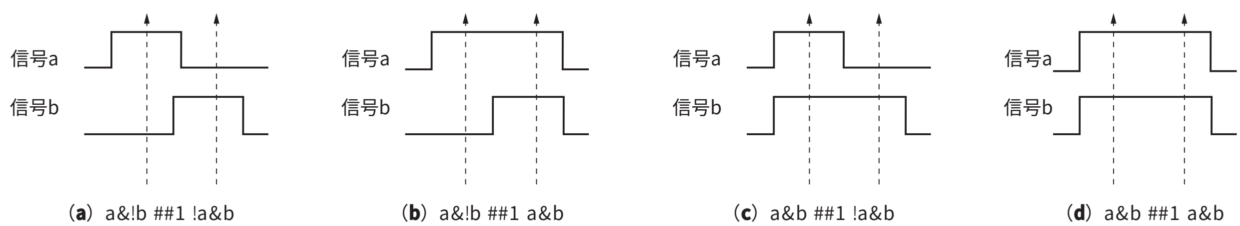


図4　a ##1 bでチェックされる四つのパターン
書いていない論理はチェックできない．この四つのパターンを区別したい場合には，個別にアサーションを記述する必要がある．

リスト2　module内に記述した並列アサーション

```
module sample(・・・);
  /* ポート，信号定義や設計など*/
  assert property (@(posedge clk)  a ##1 b );
  /* 設計 */
endmodule
```

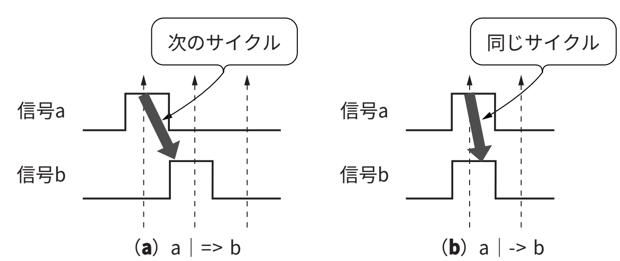


図5　|=>と|->の違い
`|=>`の場合には，`a`が'1'になった次のサイクルで`b`が'1'になる．`|->`の場合，`a`が'1'になった同じサイクルで`b`が'1'である．

き仕様があります．並列アサーションで条件を書くには含意(implication)を使用します．SystemVerilogアサーションでは二つの含意が用意されており，`|->`または`|=>`を使用します．まず，「信号`a`が'1'ならば，信号`b`が'1'」というプロパティは，以下のように記述します．

- `a |-> b`：「`a`ならば，同じサイクルで`b`が'1'になる」
- `a |=> b`：「`a`ならば，次のサイクルで`b`が'1'になる」

図5に示すように，タイミングが異なる点に注意してください．

含意を使った場合，アサーションの評価結果は3種類あります．すなわち，「条件が成り立ってPASS」，「条件が成り立ってFAIL」，「条件が成り立たずPASS」という三つの場合です．これを示したのが**図6**です．T0では，`a`が'1'で条件が満たされ，T1で`b`が'1'になり，アサーションはPASSとなります．T1では，`a`が'1'で条件が満たされますが，T1で`b`が'0'になるので，アサーションはFAILです．T2では，`a`が'0'のため，そもそも条件が満たされないので，アサーションとしてはPASSになります．

アサーションでチェックを行う場合，条件が成り立ったときのPASSとFAILが重要です．ここから先は，この「条件が成り立ったときのPASS」と「条件が成り立ったときのFAIL」をそれぞれPASS，FAILと呼ぶことにします．

● サイクル遅延で幅のあるサイクルを指定する

もう少し複雑なアサーション記述の例を紹介します．

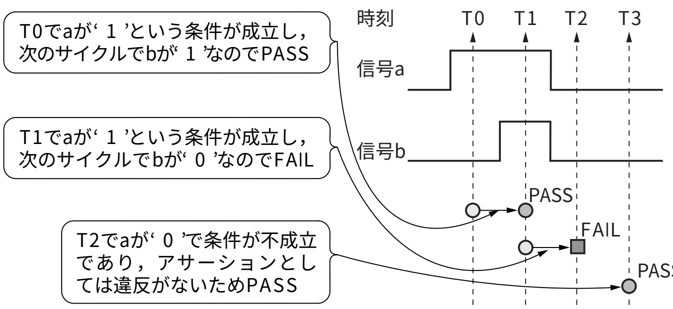


図6　assert property (@(posedge clk) a |=> b);の結果

含意(implication)を使って条件付きのアサーションを記述すると，その結果は3種類になる．

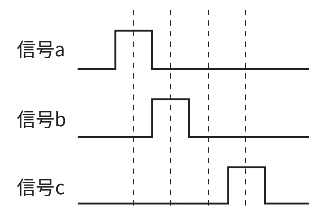


図7　assert property (@(posedge clk) a |=> b ##2 c);

「信号aが‘1’ならば，次のサイクルで信号bが‘1’，その2サイクル後にcが‘1’になる」という仕様に対応する．

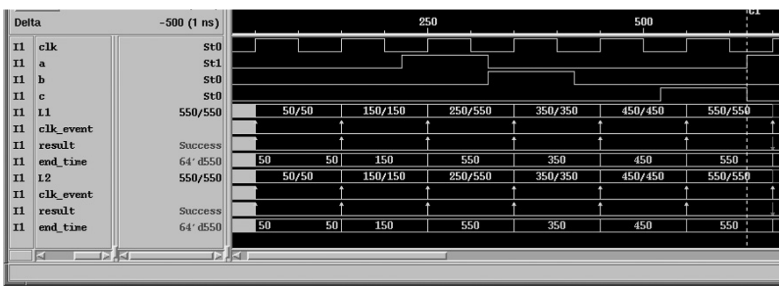


図8　VCSによる波形表示

アサーションでエラーがなければ，結果(result)が↑で示される．エラーがあれば↓で示される．

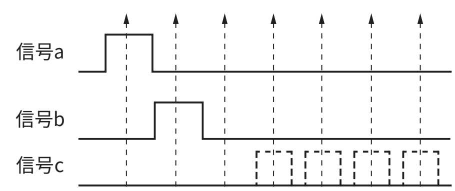


図9　assert property (@(posedge clk) a |=> b ##[2：5] c);

「信号aが‘1’ならば，次のサイクルで信号bが‘1’，その2～5サイクル後にcが‘1’になる」という仕様に対応する．

1) Nサイクル後(##N)の記述

「信号aが‘1’ならば，次のサイクルで信号bが‘1’に，その2サイクル後に信号cが‘1’になる」というアサーションを書いてみましょう(**図7**)．2サイクル後として##2を使えば，以下のように記述できます．

```
assert property ( @(posedge clk)
                              a |=> b ##2 c );
```

また，真理値の1'b1を使えば，「つねに成立している」という意味になります．実際にはチェックが不要なダミー・サイクルを用意して，「信号aが‘1’ならば，次のサイクルで信号bが‘1’，その次のサイクルはダミー・サイクルで，さらに次のサイクルに信号cが‘1’になる」として，以下のように記述することができます．

```
assert property ( @(posedge clk)
                     a |=> b ##1 1'b1 ##1 c );
```

この二つのアサーションを使って米国Synopsys社のHDLシミュレータ「VCS」でシミュレーションを行うと，**図8**のような波形を表示します．

2) N～Mサイクル後(##[N:M])の記述

回路によっては動きに幅があるものもあります．例えば，「信号aが‘1’ならば，次のサイクルで信号bが‘1’，その2～5サイクル後にcが‘1’になる」というものです(**図9**)．2～5サイクル後は##[2:5]と記述できるので，このアサーションは以下のようになります．

```
assert property ( @(posedge clk)
                         a |=> b ##[2:5] c );
```

● 連続繰り返しを利用して記述をコンパクトに

ここまでの記述でもかなりのアサーションを記述できますが，以下で紹介する繰り返し記述はアサーションをコンパクトにするためにたいへん有効なものです．

1) Nサイクル連続繰り返し([*N])の記述

例えば，「信号aが‘1’ならば，次のサイクルから100回連続でbが‘1’になる」というアサーションをこれまでの説明だけで記述しようとすると，以下のようになります．

```
assert property ( @(posedge clk)
           a |=> b ##1 b ##1 .... ##1 b );
```

ここでは誌面のつごう上省略していますが，このようなアサーションをまじめに書いてはいけません．このアサーションが正しいかどうかをチェックするのがたいへんになります．SystemVerilogでは連続繰り返しを行うための記述法が用意されています．b[*100]と記述することで，「1ビットの信号bは連続100サイクルで‘1’である」という意味になります．記述を書き換えてみると，以下のようにたいへん短く書くことができます．

```
assert property ( @(posedge clk)
                      a |=> b[*100] );
```

2）N～Mサイクル連続で繰り返し（[*N ： M]）の記述

さらに連続繰り返しでは，繰り返し回数に範囲を持たせた応答を記述することができます．例えば「信号aが‘1’ならば，次のサイクルから2～4回連続でbが‘1’になる」というアサーションを記述できます（**図10**）．b[*2:4]と書くと「2～4回連続でbが‘1’になる」という意味になるので，アサーションは以下のように記述できます．

```
assert property ( @(posedge clk)
                      a |=> b[*2:4] );
```

● シミュレーション終了時まで続く場合は$を使う

また，「シミュレーションが終了するまで続く」と指定したい場合があります．しかし，シミュレーションの終了時間はテストベンチによって異なりますし，強制的に終了されることもあるかもしれません．SystemVerilogではそのような条件を $ を使って表現します．

```
a[*10:$]
// 10からシュミレーション終了までに1ビットの信号aが‘1’
```

もう少し大きな範囲で繰り返しを指定することもできます．「『1ビットの信号aが‘1’で，次のサイクルに1ビットの信号bが‘1’』を10回繰り返す」という部分の記述を以下に示します．

```
(a ##1  b)[*10]
```

こうすることで，かなり複雑な記述でもコンパクトに書くことができます．

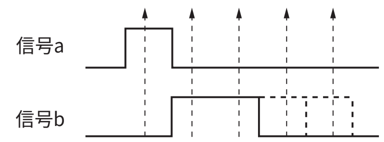


図10　assert property (@ (posedge clk) a |=> b [*2 ： 4]);
「信号aが‘1’ならば，次のサイクルから2～4回連続でbが‘1’になる」という仕様に対応する．

● 連続ではない繰り返しの記述法は2種類ある

次に，途中でとぎれてもよいという条件で，100回信号aが‘1’になるというアサーションを作ろうとすると，これまでの記述では書きにくいことがわかります．SystemVerilogでは連続ではない繰り返しを記述するために，2種類の記述をサポートしています．

- x[->100]，x[=100]：連続または非連続で100回信号xが‘1’になる．
- x[->2:5]，x[=2:5]：連続または非連続で2～5回信号xが‘1’になる

この->と=の違いを見てみます．b[->2] ##1 cは，「2回目にbが‘1’になった次のサイクルでcが‘1’」というプロパティになりますが，b[=2] ##1 cは，「2回目にbが‘1’になってから，いつかcが‘1’になる」というプロパティになります．この[->2]をGOTO繰り返しと呼び，[=2]を非連続繰り返しと呼びます．

GOTO繰り返しの重要な点を，a |=> b[->2] ##1 cの例を使って**図11**で説明します．一つ目のポイントは，GOTO繰り返しb[->2]とその前の信号aの成立が隣り合っている必要がないことです．二つ目のポイントは，b[->2]は連続しても非連続でもよいことです．三つ目のポイントは，b[->2]とcが連続していなければならないことです．

非連続繰り返しの場合，GOTO繰り返しの一つ目と二つ目のポイントは同じですが，**図12**に示すようにb[=2] ##1 cが連続しても非連続でもよいことが違いとなります．この三つの繰り返しの動きを**図13**に示します．GOTO繰り返しと非連続繰り返しの四つのポイントを注意していただければ理解しやすいと思います．

SystemVerilogの連続繰り返し，GOTO繰り返し，非連続繰り返しの三つの使い分けは難しく感じるかもしれませ

5

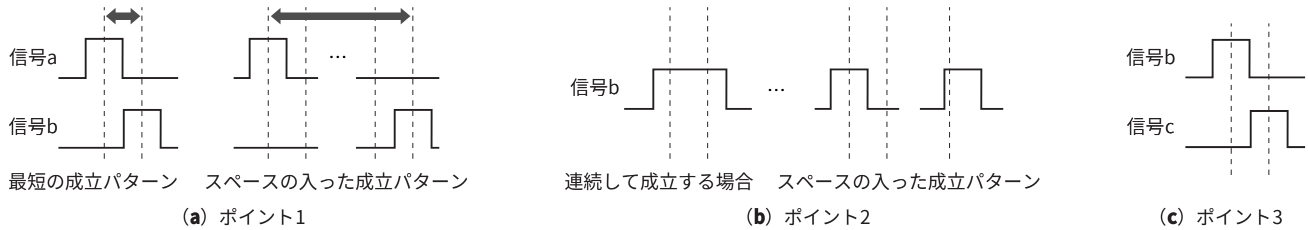


図11　GOTO繰り返し(a |=> b [->2] ##1 c)の動作

(a)のように，GOTO繰り返しの前に非成立区間があってよい．また，`a`と`b`の成立の間にスペースがいくつあってもよい．
(b)のように，GOTO繰り返しは連続していても，離れてもよい．また，`b`が連続2回成立しても，ばらばらに2回成立してもよい．
(c)のように，GOTO繰り返しと後続の`c`は，連続してはならない．また，2回目の`b`が成立したら，次のサイクルで`c`が成立しなければならない．

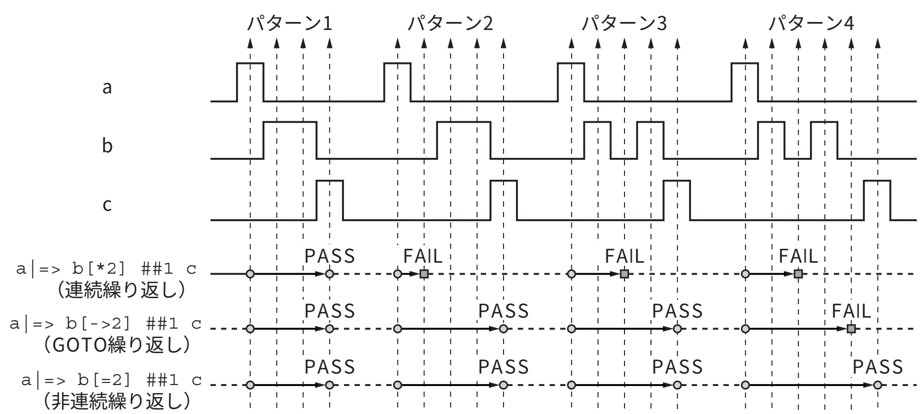


図13
繰り返し記述の違いによる動作の違い

連続繰り返し，GOTO繰り返し，非連続繰り返しの動作の違いを示した．

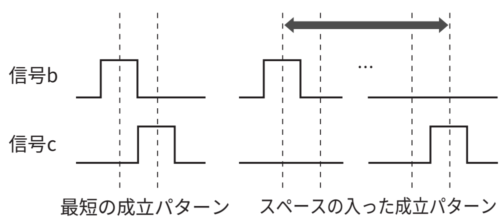


図12　非連続繰り返し(a |=> b [=>2] ##1 c)の動作

GOTO繰り返しとの違いは，後続との関係にある．GOTO繰り返しのポイント1とポイント2は，非連続繰り返しでも同じ．`a |=> b [=2] ##1 c`の非連続繰り返しと後続の`c`は，連続していても，離れていてもよい．離れていてもよい点が，GOTO繰り返しの場合と異なる．

ん．しかし，どのようなプロパティを書くのかで使い分けることになるので，慣れてくるとそれほど迷わずに選ぶことができます．

● disable iffでアサーションをキャンセル

アサーションで困ることの一つに，回路が正しい動作なのにまちがった動作と判定されることがあります．このような現象はリセット動作などによって発生します．

図14(a)の3ビット・カウンタを例に説明します．このカウンタの検証時に，「countが2ならば，次のサイクルでcountは3で，その次のサイクルでcountは4である」というアサーションでチェックすることにしました．このアサーションは以下のように書けます．

```
assert property ( @(posedge clk)
  count==3'd2 |=> count==3'd3
                         ##1 count==3'd4 );
```

通常の動作ではこれは正しく動くように感じますが，このカウンタにはリセット入力があるので，リセット時にこのアサーションはFAIL(バグ発見)になってしまいます．しかし，リセット動作は正しい動作の一部なので，FAILにはしたくありません．

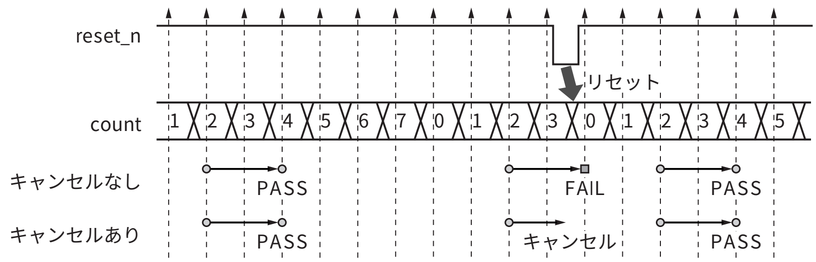
このような誤った判定を防ぐため，SystemVerilogアサーションでは`disable iff`という構文があります．

```
assert property( @(posedge clk)
  disable iff (!reset_n)
  count==3'd2 |=> count==3'd3
                         ##1 count==3'd4 );
```

この`disable iff (!reset_n)`を入れることで，**図**

```
always @(posedge clk or negedge reset_n)
  if(!reset_n) count<=3'd0;
  else         count<=count+1'b1;
```

(a) 検証対象の3ビットのカウンタ



(b) 回路の動作とアサーションの動き

図14　アサーションをキャンセルする例

FF(フリップフロップ)を含む回路では，リセットのようにアサーションをキャンセルしなければならない場合がある．アサーションを記述するときに，リセットなどの条件をつねに考える癖をつけるとよい．

14(b)のようにreset_nが'0'のときにアサーションの条件が成立しないことになります．これにより，リセット時などの挙動をアサーションで表現することができます．

● property宣言を使ってassertを分離する

これまでは，1行でアサーションを書いていましたが，記述の効率化や再利用などを考えて，propertyを宣言し，それをassert propertyで指定するという記述も行えます(**リスト3(a)**)．これだけでは一つの記述を二つに分離しただけですが，propertyではさらに引き数を使用することができます．Verilog HDLのポートの定義と同じように，プロパティ名pABの後にかっこを付けて引き数をコンマで区切って定義すると，その引き数でプロパティを定義できます．引き数を利用すると，アサーションの再利用性が向上します．例えば，**リスト3(a)**の記述では信号a，bを変えることができませんが，これを引き数x，yを使って定義しておく(**リスト3(b)**)ことで，ほかの信号でも利用可能なプロパティとなります．

このように引き数を使って仮の名まえでアサーションを記述しておくと，インスタンスするときに実際にチェックしたい信号を指定することができます．インスタンスの方法は，Verilog HDLと同じようにポート名で指定する方法と順番で指定する方法を利用できます．それぞれ，以下のような記述になります．

```
assert property (pAB2( .x(a), .y(b)) );
assert property (pAB2(a,b));
```

● 任意のシーケンスに名まえを付ける

アサーション記述で注意したほうがよいのは，同じ記述をいろいろな場所に別々に書かないようにすることです．

リスト3　property宣言とアサーション定義

```
property pAB;  // プロパティpABの定義
  @(posedge clk)
  a  |=>  b;
endproperty
assert property (pAB); // pABを使ったアサーション定義
```

(a) propertyで宣言

```
property pAB2(x,y); // 引き数を持ったプロパティpAB2の定義
  @(posedge clk)
  x  |=>  y;
endproperty
```

(b) 引き数を使って定義

理由は，同じ記述をいろいろな場所に書いていると，仕様の変更が発生した場合に，すべてのアサーションを修正する必要が出てくるからです．このためにはpropertyを使用したり，以下で紹介するsequenceを使い，一つの仕様を1ヵ所に記述することがポイントとなります．

シーケンス定義によってプロパティを記述するための部品を作成することができますが，含意(|->，|=>)を使用できない点がpropertyの場合と異なります．シーケンス定義は以下のように行います．

```
sequence  <名まえ>;
   <定義したい条件>;
endsequence
```

例えば，「信号aが'1'になって，次のサイクルで信号bが'1'になる」というシーケンスは，以下のように定義できます．

```
sequence  qAB ;
   a ##1 b;
endsequence
```

表1　シーケンスの演算

機　能	記　述	意　味
シーケンスの連結	S1 ##1 S2	S1が終わった次のサイクルでS2がスタートするシーケンス
シーケンスの融合	S1 ##0 S2	S1の最後のサイクルでS2がスタートするシーケンス
シーケンスのor	S1 or S2	S1またはS2のどちらかが成立するシーケンス
シーケンスの（長さ不一致）and	S1 and S2	S1とS2の両方が成立するシーケンス
シーケンスの長さ一致and	S1 intersect S2	長さが同じS1とS2が両方成立するシーケンス

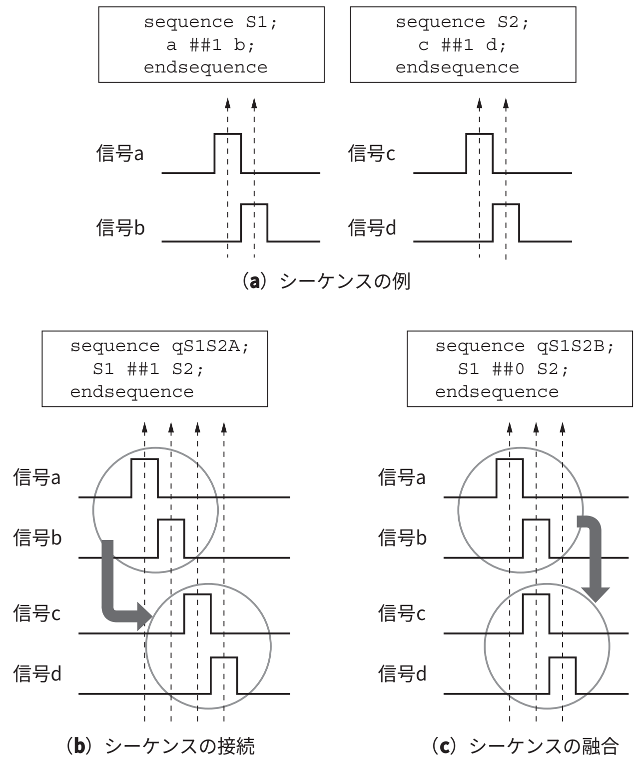


図15　シーケンスの連結
シーケンスを接続するときに##1を使用する．また，##0を使うことで，オーバラップ（融合）させることもできる．

一度定義したシーケンスは，以下のように名まえを呼び出すことで使用できます．

```
property pAB_C;
  @(posedge clk) qAB |=> c;
endproperty
```

シーケンスを使うと，一つの記述を複数のプロパティで利用できます．これにより，仕様変更などに対応しやすくなります．

● **シーケンスでは引き数を利用できる**

プロパティの場合と同じように，シーケンスでも引き数を利用できます．呼び出すときに実際の信号を引き数で指定できるので，より多くの記述に対応できます．Verilog HDLのポート定義と同じように定義することができます．

```
sequence  qAB2(x,y);
  x ##1 y;
endsequence
```

使用する際も，呼び出すときに引き数に実際の信号を与えることで利用できます．

```
property pAB_C2;
  @(posedge clk) qAB2(a,b) |=> c;
endproperty
```

引き数を用いたシーケンス定義を使うと，より再利用性が高まります．これはぜひ使っていきたい機能です．

● **シーケンスどうしの演算が定義されている**

シーケンスの定義や呼び出しができるようになったら，これらとシーケンスの演算を組み合わせることで，さらに効率的に記述できるようになります．今回は，**表1**に挙げた比較的使用頻度の高いものについて説明します．ここでは，チェックしたい波形をどう記述するのかをイメージするとよいと思います．

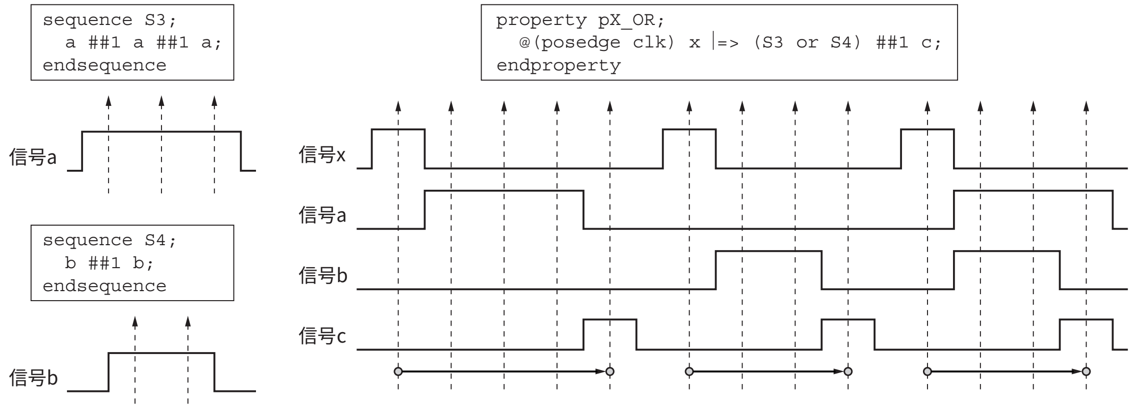
1）シーケンスの連結（##1）

ある二つのシーケンスを接続すると，簡単にアサーションを記述できることがあります．そういった場合，1サイクルの遅延（##1）を使うことで容易に記述できます．例えば，**図15**のようなシーケンスS1とS2がある場合，それらを接続したシーケンスは以下のように記述できます．

```
sequence qS1S2A;
  S1 ##1 S2;
endsequence
```

2）シーケンスの融合（##0）

また，このS1の最後のサイクルとS2の最初のサイクルをオーバラップ（融合）したい場合，0サイクルの遅延（##0）を使うことで，以下のように記述できます．



(**a**) シーケンスの例　　(**b**) シーケンスのOR

図16 シーケンスのor
二つのシーケンスのどちらかが成立するアサーションを記述する場合にorを使用する．両方成立する場合には，早く成立したほうが選択される．

```
sequence qS1S2B;
  S1 ##0 S2;
endsequence
```

3）シーケンスのor

シーケンスのorでは，「二つのシーケンスのうちのどちらか一方が成立したら」という記述を行うことができます．**図16**の(S3 or S4) ##1 cのアサーションが成立するのは，S3が成立し，次のサイクルで信号cが1'の場合と，S4が成立し，次のサイクルで信号cが1'の場合があります．シーケンスのorと言っているので，S3とS4が同時に成り立つ場合もありますが，その場合は最初に成り立ったシーケンスで評価を行います．**図16**の場合はS4のシーケンスが短いため，S4 ##1 cで評価を行うことになります．

4）シーケンスのand

シーケンスの長さが異なっていて，「二つのシーケンスがともに成り立つ」という記述を行いたい場合，andを使用します．シーケンスの長さとは，そのシーケンスをチェックするのに必要なサイクルです．**図17**のS3のシーケンスの長さは3サイクル，S4は2サイクルです．このシーケンスのandをとった場合，両方のシーケンスが成立した時点が完了ポイントとなります．ただし，成立する場所はシーケンスの長いほうになるので，後続のcを評価するタイミングに注意が必要です．

5）シーケンスの長さ一致and（intersect）

ある二つのシーケンスが同時に発生し，同時に終了する必要がある場合があります．そのような場合，intersectを使用します．先ほどのS1とS2を使ったプロパティの例を以下に示します．S1とS2の長さが一致するものだけが成立することに注意してください．

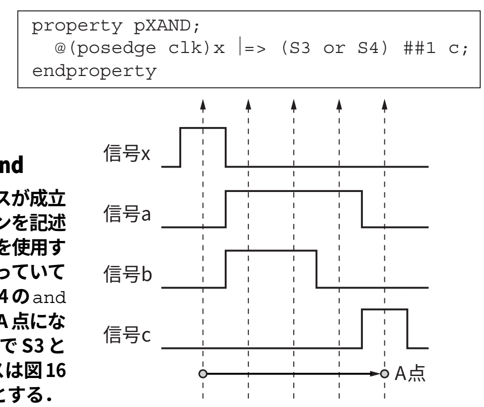


図17 シーケンスのand
二つのシーケンスが成立するアサーションを記述する場合にandを使用する．長さが異なっていてもよく，S3とS4のandが成立するのはA点になる．なお，ここでS3とS4のシーケンスは図16(a)と同じものとする．

```
property pX_S1S2;
  @(posedge clk)
  x |=>  S1 intersect S2;
endproperty
```

（後編に続く）

あかぼし・ひろき
(株)ロジック・リサーチ

<著者プロフィール>
赤星博輝．ロジック・リサーチに所属しており，ハードウェアからソフトウェアまで担当するなんでも屋．もともとはEDAツールの開発を行っていたが，現在はハードウェア設計から徐々に検証技術に移っているところ．今後，どのようなしごとをするのかは，世の中の進みかたしだいです．

Column SystemVerilogの情報源

Design Wave Magazine 編集部

ここではSystemVerilog関連の情報が掲載された記事や書籍，ホームページなどをご紹介いたします．

● SystemVerilog関連記事

- **「記述能力，再利用性，検証機能を強化した次期Verilog言語の全ぼう——SystemVerilogの特徴と評価結果」**
  赤星博輝 著，Design Wave Magazine，2004年3月号，pp.96-105.
- **「SystemVerilogで簡易CPUバス・モデルを記述——記述を効率化し，ミスを少なくするVerilog HDL言語の進化型」**
  宮下晴信 著，Design Wave Magazine，2004年8月号，pp.80-90.
- **「続・SystemVerilogで簡易CPUバス・モデルを記述——DPIを使ってC++モデルを接続する」**
  宮下晴信 著，Design Wave Magazine，2004年12月号，pp.121-129.
- **「クラスの概念を利用してSystemVerilogモデルの再利用性を向上——簡易CPUバス・モデルのクラス記述」**
  宮下晴信 著，Design Wave Magazine，2005年5月号，pp.106-113.
- **書評「SystemVerilogによるLSI設計」，「SystemVerilog Assertions Handbook」**
  赤星博輝 著，Design Wave Magazine，2005年6月号，p.58.

● 検証言語，アサーション，プロパティ検証関連記事

- **「特集 デバッグ鉄人への道，第7章『検証言語』導入体験記——Intel社での経験が効率的な検証にこだわるきっかけに」**
  森田年一 著，Design Wave Magazine，1999年9月号，pp.66-73.
- **「特集1 SOC開発プロジェクトを成功させるには…，第4章 システムLSIの機能検証——検証方法を体系的に考える検証エンジニアの出番」**
  大上晃弘，北沢倫子 著，Design Wave Magazine，2002年8月号，pp.60-65.
- **「HDLシミュレータのデバッグ機能を使いこなす——波形表示，カバレッジ測定，アサーション記述」**
  伊達啓次郎，富田孝造 著，Design Wave Magazine，2004年1月号，pp.93-97.
- **「特集1 テスト・クライシスの処方せん，第2章 LSI検証クライシスの『傾向と対策』——ソフトウェア技術を身につけた検証チームの導入で検証問題を解消」**
  宮原忠義，難波 睦 著，Design Wave Magazine，2004年2月号，pp.30-39.
- **「アサーションを活用して不ぐあい箇所を即座に特定（前編）——市販HDLシミュレータのサポートが始まった新デバッグ手法」**
  赤星博輝 著，Design Wave Magazine，2004年8月号，pp.129-134.
- **「アサーションを活用して不ぐあい箇所を即座に特定（中編）——複数サイクルにまたがる仕様をPSL/Sugarでチェック」**
  赤星博輝 著，Design Wave Magazine，2004年12月号，pp.130-135.
- **「アサーションを活用して不ぐあい箇所を即座に特定（後編）——複雑なシーケンスを簡潔に表現するテクニック」**
  赤星博輝 著，Design Wave Magazine，2005年1月号，pp.99-106.
- **「連載・システムLSI検証技術入門，第9回 論理回路のプロパティ検証技術——順序回路に適用するフォーマル・ベリフィケーション」**
  藤田昌宏 著，Design Wave Magazine，2004年8月号，pp.135-141.
- **「連載・システムLSI検証技術入門，第10回 論理回路のプロパティ検証技術（2）——仕様との間に矛盾がないことをチェックする方法」**
  藤田昌宏 著，Design Wave Magazine，2004年10月号，pp.136-141.
- **「連載・システムLSI検証技術入門，第11回 論理回路のプロパティ検証技術（3）——プロパティ検証の実際」**
  藤田昌宏 著，Design Wave Magazine，2005年2月号，pp.144-148.
- **「連載・システムLSI検証技術入門，第12回 論理回路のプロパティ検証技術（4）——時相論理による仕様記述法」**
  藤田昌宏 著，Design Wave Magazine，2005年7月号，pp.144-148.
- **「連載・システムLSI検証技術入門，第13回 論理回路のプロパティ検証技術（5）——プロパティ記述言語（PSL）の構文と使いかた」**
  藤田昌宏 著，Design Wave Magazine，2005年9月号，pp.144-149.

● SystemVerilog関連書籍

- **SystemVerilog Golden Reference Guide v3（supporting System Verilog 3.1a）**
  Doulos 刊，ISBN：0-9547345-0-5，20.6×12cm，468ページ，50ドル，2003年3月（初版発行）.
- **SystemVerilogによるLSI設計**
  Stuart Sutherland，Simon Davidmann，Peter Flake 著，浜口加寿美，河原林政道，高嶺美夫，明石貴昭 訳，丸善 刊，ISBN：4-621-07515-2，A5判，338ページ，4725円，2005年1月.
- **SystemVerilog Assertions Handbook**
  Ben Cohen，Srinivasan Venkataramanan，Ajeetha Kumari 著，Vhdlcohen Publishing 刊，ISBN：0-9705394-7-9，21.3×27.5cm，366ページ，150ドル，2005年.
- **A Practical Guide for SystemVerilog Assertions**
  Srikanth Vijayaraghavan，Meyyappan Ramanathan 著，Springer 刊，ISBN：0-387-26049-8，334ページ，125ドル，2005年6月.

● SystemVerilog関連ホームページ

- **Accelleraのホームページ**
  http://www.accellera.org/
- **SystemVerilogのホームページ**
  http://www.systemverilog.org/
- **SystemVerilogのEDAベンダによるサポート状況**
  http://www.systemverilog.org/products/products_solu.html
- **SystemVerilog 3.1a Language Reference Manual（LRM）**
  http://www.eda.org/sv/SystemVerilog_3.1a.pdf